САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
010500 «МАТЕМАТИЧЕСКОЕ ОБЕСПЕЧЕНИЕ И
АДМИНИСТРИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ»
ПРОФИЛЬ: «ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ»

Веселов Александр Алексеевич

Масштабирование веб-приложений

Бакалаврская работа

Научный руководитель:
ст. преп. Кузьменко В. Г.

Рецензент:
к. ф.-м. н., доц. Лебединский Д. М.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

010500 “SOFTWARE AND ADMINISTRATION
OF INFORMATION SYSTEMS”

SPECIALIZATION: “TECHNOLOGIES OF PROGRAMMING”

Aleksandr Veselov

Web-application scalability

Bachelor’s thesis

Supervisor:

Senior Lecturer Victor Kuzmenko

Reviewer:

Associate Professor, Candidate of
Physico-Mathematical Sciences
Dmitri Lebedinski

Saint-Petersburg

2016

Оглавление

Введение

На сегодняшний день совершенно очевидно, что с каждым годом число пользователей Интернета возрастает. В то же время, распространение сервисов потоковой передачи данных, социальных сетей, появление концепции Интернета вещей [1] (Internet of Things, IoT) лишь увеличило объем передаваемой информации. В современных условиях особо остро встаёт вопрос построения систем, способных обрабатывать сотни тысяч запросов в минуту и обладающих высокой степенью доступности. В то же время, затраты на разработку и развёртывание систем, ориентированных на 100, 10 000, 1 000 000 и 100 000 000 пользователей существенно различаются. Если в первом случае достаточно использовать для развёртывания веб-приложения бесплатный хостинг или недорогую виртуальную машину на одной из облачных платформ, то для работы таких огромных систем, как Youtube или Facebook, требуются многочисленные центры обработки данных по всему миру. Разработка и поддержка высокопроизводительной системы может не окупиться, если число её пользователей будет существенно меньше запланированного.

Аналогичным образом, различаются и подходы к реализации систем. Методы, применяемые при создании небольших приложений, ориентированных на небольшое число пользователей, например, интернет-магазина, могут быть неприменимы к созданию глобальных вычислительных систем. В то же время, технологии промышленной разработки и, в особенности, методы оптимизации, применяемые в разработке крупнейших веб-приложений, порождают избыточную сложность при создании малых приложений, и даже могут ухудшить их работу.

В связи с вышеперечисленными проблемами встаёт вопрос масштабирования веб-приложения, то есть изменения его производительности в зависимости от нагрузки. Различают вертикальное и горизонтальное направления масштабирования.

Вертикальное масштабирование. Данное направление предусматривает увеличение вычислительной мощности среды запуска приложения. В связи с высокой стоимостью высокопроизводительных серверов, техническими и временными ограничениями данный способ имеет ограниченную применимость. Тем не менее он может особенно успешно применяться в облачных средах, где вычислительную мощь среды можно изменить с минимальными временными и финансовыми затратами.

Горизонтальное масштабирование. Этот способ заключается в увеличении числа запущенных экземпляров приложения и распределении нагрузки между ними. Вместе с тем, архитектура приложения значительно усложняется, требуются дополнительные затраты на поддержание работы системы, а также необходимо внедрение дополнительных компонентов приложения, таких как служба балансировки нагрузки. Тем не менее, данный способ является основным методом повышения отказоустойчивости.

Таким образом, для создания масштабируемых приложений требуются особые архитектурные и технологические решения, способные обеспечить горизонтальную и вертикальную масштабируемость.

1. Подходы к созданию масштабируемых веб-приложений

Одним из важнейших аспектов архитектуры высоконагруженного приложения является масштабируемость, то есть способность приложения увеличивать производительность при привлечении дополнительных ресурсов. Для решения задачи построения высокомасштабируемых приложений применяются разнообразные алгоритмы и архитектурные решения, языковые средства и среды запуска приложения. Тем не менее, не существует единого стандартного решения задачи масштабирования, каждая система требует собственного подхода. Выбор средств разработки масштабируемых приложений определяется исходя из типа и объема обрабатываемых данных, применяемых алгоритмов, частоты использования тех или иных функций и других факторов.

1. Специфика масштабирования веб-приложений

Как и многие другие методы оптимизации программ, вертикальное масштабирование опирается на специфику предметной области приложения. В случае веб-приложения можно выделить некоторые существенные для масштабирования особенности:

- приложение ориентировано на выполнение большого числа одновременных запросов (как правило, больше количества вычислительных ядер/процессоров в системе);
- запросы могут исполняться за существенно различное время. В то же время задержка исполнения других запросов из-за одного, время исполнения которого велико, недопустима;
- выполнение запроса может быть приостановлено в связи с ограниченной пропускной способностью сети и/или задержками при чтении/записи данных. Эти ситуации должны обрабатываться во избежание простоя системы;

- возникает необходимость в долгоживущих процессах, например, для поддержания постоянного соединения или передачи значительного объема данных. Обилие подобных процессов при неправильной архитектуре способно парализовать работу системы;
- аварийные ситуации при исполнении запроса не должны прерывать исполнение других запросов.

По этой причине, масштабирование веб-приложения должно быть учтено на стадии разработки архитектуры и выборе средств разработки.

2. Типовые архитектуры веб-приложений и их масштабируемость.

Рассмотрим некоторые стандартные архитектуры веб-приложений с точки зрения трудоемкости разработки и масштабируемости.

Наиболее простой архитектурой приложения является *монолитное приложение*. Эта архитектура подразумевает размещение всей логики и данных приложения на одном сервере и является наиболее простой в разработке и подходящей для малых приложений. Помимо этого, к достоинствам системы можно отнести высокую скорость доступа к данным. Однако, масштабируемость данного типа приложений невысока и ограничивается созданием точных копий приложения и обеспечением балансировки нагрузки, напр. с помощью HAProxy. Данная схема масштабирования может успешно применяться в случае, когда нагрузка распределяется по всем единицам функциональности равномерно и не требуется синхронизация данных между серверами. Хорошим примером приложения, для которого данная схема хорошо походит может служить онлайн-версия шахмат.

В ситуациях, когда данных становится много и требуется их синхронизация, разумно использовать *архитектуру с выделенной БД*. Данная архитек-

тура подразумевает размещение базы данных на отдельном сервере и обеспечение доступа к ней серверов логики приложения. Это позволяет разделить данные и логику, тем самым исключить потерю доступа к данным при отказе сервера логики приложения. При этом сервер базы данных становится уязвимым местом приложения, его отказ влечет за собой отказ приложения, если требуется согласованность данных. Однако, если согласованность не обязательна или может нарушаться в определенные моменты времени, данная архитектура является оптимальным вариантом, так как является основой для более сложных схем масштабирования. Впоследствии возможно разделение БД на меньшие части и/или увеличение надежности и доступности данных путем репликации. Тем не менее, если функционал приложения задействуется неравномерно, данная архитектура не является эффективной с позиций использования вычислительных мощностей.

На сегодняшний день особый интерес представляют сервис-ориентированные архитектуры (SOA), и, как частный случай SOA, архитектура микросервисов [2]. Идея микросервисов заключается в дроблении логики приложения на мелкие фрагменты, каждый из которых реализует свой фрагмент функциональности приложения и использует свои данные. Это позволяет более гибко организовывать доступ к данным и управлять нагрузкой. Например, для реализации функциональности социальной сети можно использовать микросервис, обрабатывающий сообщения, микросервис постов (с документоориентированной БД), сервис учетных данных (с key-value хранилищем), службу видеозаписей (специальные сервера с мощными GPU и высокопроизводительные дисковые хранилища). Также, архитектура микросервисов позволяет эффективно организовать коллектив разработчиков, разбив его на команды, каждая из которых занимается разработкой собственного микросервиса независимо от других. И еще микрослужбы позволяют разрабатывать приложения с использованием различных языков программирования и для различных платформ, что повышает производительность труда программиста за счет использования средств, наиболее подходящих для решения конкретной задачи. С

точки зрения масштабируемости, микросервисы лучше всего поддаются горизонтальному масштабированию. Возможно не только использование отдельного сервера для каждого сервиса, но и совместное размещение микросервисов на одной физической или виртуальной машине. Возможно гибкое управление нагрузкой, например, создание новых экземпляров микросервиса ко времени предполагаемого роста нагрузки и удаление, после спада. Из недостатков следует отметить то, что микросервисы создают высокие накладные расходы, и их применение оправдано только в больших приложениях.

Так же с микросервисами связан подход к предоставлению вычислительных мощностей, заключающийся в размещении микросервисов в едином глобальном пространстве, вместо предоставления виртуальных и/или физических серверов. На данный момент существуют 2 реализации данного подхода в облачных платформах: AWS Lambda от Amazon и Azure Service Fabric от Microsoft. Первая платформа ориентирована на создание микросервисов-обработчиков событий, происходящих в среде AWS. Этот подход [3] удобен для выполнения служебных задач и асинхронной обработки данных. В свою очередь Service Fabric предлагает построение классических микросервисов различных моделей и обеспечение их взаимодействия с помощью «надежных коллекций» [4].

3. Средства языка программирования Go для разработки масштабируемых веб-приложений

Хорошим примером языковых технологий, ориентированных на создание масштабируемых приложений является язык Go [5], разработанный в 2009 г. Ключевой особенностью данного языка является реализация идей языка CSP [6], предложенного Т. Хоаром в 1978 г., в частности идеи процессов, общающихся между собой при помощи каналов [7]. В Go подобные процессы получили название горутин, которые обеспечивают параллельную обработку данных. В начале работы программы среда исполнения Go создает N потоков, где

N – число процессоров/ядер системы. Затем в процессе работы программы создаются горутины, которые среда исполнения распределяет по потокам таким образом, чтобы обеспечить максимально эффективную загрузку потоков и равноправие горутин. Важным моментом является то, что во время работы приложения могут одновременно существовать тысячи горутин, что отличает их от обычных потоков.

Механизм каналов позволяет решить многие проблемы, связанные с синхронизацией. Так, горутины и каналы очень просто описывают конвейерную обработку данных: каждой операции соответствует множество горутин одного вида, каналы же осуществляют передачу данных от одного этапа вычисления к другому. Важным моментом является то, что, во-первых, канал может использоваться совместно несколькими горутинами (при этом канал потокобезопасен), и, вовторых, каналы сами по себе являются объектами и могут быть переданы горутинам, например, для предоставления непосредственного (как частный случай – монопольного) доступа к процессу (напр. осуществляющему доступ к принтеру).

2. Описание приложения «KtoZa»

В качестве масштабируемого веб-приложения была создана платформа для проведения опросов населения «KtoZa». Потребность в данного рода приложениях продиктована необходимостью проведения опросов в кратчайшие сроки с привлечением наибольшего числа респондентов. Данное приложение позволяет создать и провести опрос населения с меньшими финансовыми и временными затратами. К данному приложению предъявляются следующие требования:

- возможность управления опросом и статистикой в графическом интерактивном режиме;
- учет числа участников опроса и числа ответов;
- отображение статистики в реальном времени для создателя опроса (наиболее приближенном к реальному);
- поддержка вопросов с одним или несколькими вариантами ответа;
- возможность запуска и остановки опроса по расписанию;
- возможность использования различных клиентских частей.

Анализ данных требований позволил сформулировать требования к реализации приложения:

- неизвестна нагрузка на приложение (определяется задачами пользователя системы), что влечет необходимость создания высокомасштабируемой системы;
- требование реального времени влечет за собой минимизацию задержек в работе приложения;
- использование системы должно быть экономически оправданным, чего можно достичь в т. ч. уменьшением накладных расходов при работе системы и возможностью работы в облачной среде (время ра-

боты приложения мало, поэтому аренда виртуального сервера в облаке представляется более выгодной по сравнению с покупкой/арендой физического сервера);

- возможность подключения внешних клиентов требует создания API серверной части, осуществляющего представление данных с помощью одного из стандартных протоколов сериализации.

3. Архитектура приложения «KtoZa»

С учетом приведенных выше требований была создана архитектура приложения, представленная на рис. 1. Приложение основано на архитектуре «клиент-сервер», но имеет некоторые существенные для масштабирования особенности. Главным аспектом архитектуры является разделение функциональности приложения для создателя опроса и для респондента. Сервер создателя опроса (M-сервер) запускается в единственном экземпляре и осуществляет операции по созданию/редактированию опроса и сбору данных с серверов респондентов. Сервер респондента (R-сервер) производит прием и агрегацию ответов респондентов с последующей отправкой промежуточных результатов M-серверу.

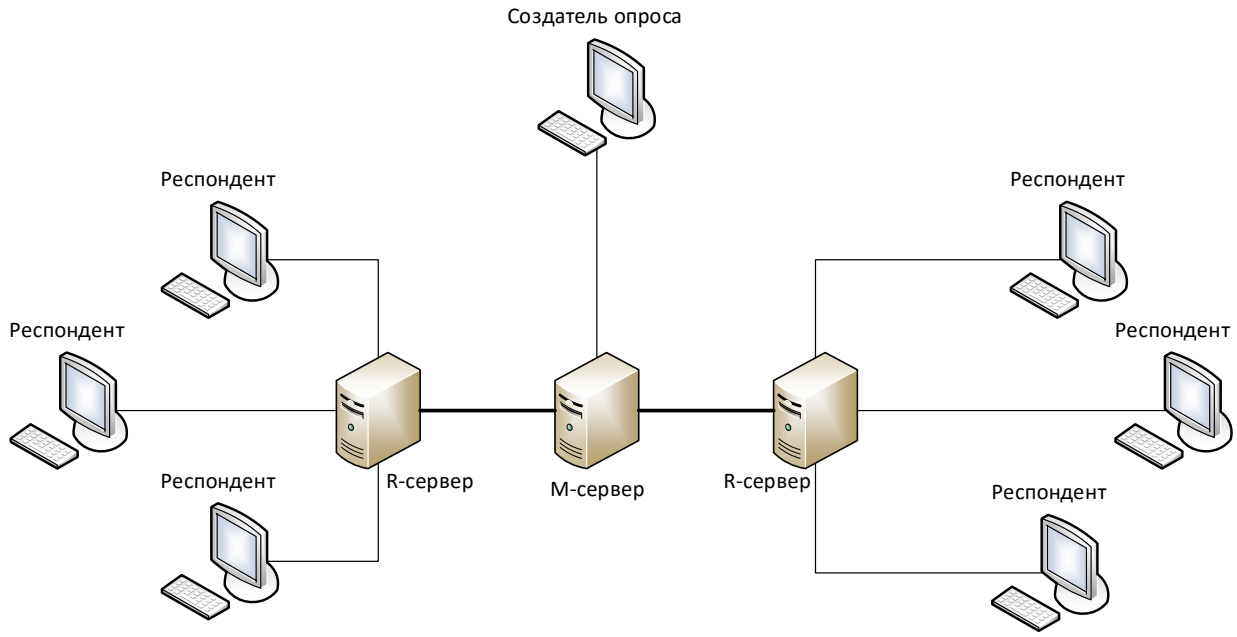

Рисунок 1. Архитектура приложения

В терминах, используемых в теореме CAP [8], данная система является AP-системой [9] ввиду требований, предъявляемых к ней. Так как требование терпимости к разделению следует из необходимости масштабирования, а требование доступности – из необходимости минимизации временных задержек, в соответствии с вышеозначенной теоремой согласованность системы в каждый момент времени не гарантируется.

Рассмотрим процесс взаимодействия серверов более подробно. Сначала запускается M-сервер, который загружает сохраненные данные (при наличии). После этого возможен запуск экземпляров R-серверов, которые устанавливают соединение и получают данные от M-сервера. Во время опроса R-сервер принимает ответы и создает из них кэш. Данный кэш асинхронно отправляется M-серверу и обнуляется. Если во время отправки происходит ошибка, отправляемая копия кэша объединяется с текущей, чтобы избежать потери результатов и производятся попытки повторного установления соединения. Таким образом возможна работа R-сервера по сбору результатов в случае отказа M-сервера. В свою очередь M-сервер принимает кэш вопросов, объединяет его с текущей статистикой, сохраняет статистику на диск и рассылает её всем подключенным R-серверам и клиенту создателя опроса.

Данная схема позволяет добиться:

1. *Высокой масштабируемости.* В силу того, что основная нагрузка ложится на R-сервера, управление нагрузкой сводится к изменению количества R-серверов. Возможно добавление R-серверов по ходу работы, но до момента начала приема ответов (число активных пользователей к началу приема ответов известно, так как ответы от пользователей, подключившихся после начала приема ответов не обрабатываются). Это позволяет более гибко управлять накладными расходами, например, использовать минимальное число R-серверов в момент создания опроса и увеличить их количество ко времени начала опроса.
2. *Отказоустойчивости.* Потеря одного R-сервера ведет к обрыву всех его текущих соединений, но так как синхронизация данных с M-сервером происходит через малые промежутки времени (около 1 с. или меньше), то утеряны будут только несинхронизированные данные. В случае отказа M-сервера данные не теряются, так как результаты продолжают поступать в кэши R-серверов. После вос-

становления работоспособности М-сервера происходит синхронизация. Также в целях обеспечения надежности происходит запись на диск статистики каждый раз, когда М-сервер получает кэш ответов и применяет его.

3. *Высокой доступности.* Хранение кэшей ответов и снимков текущей статистики в оперативной памяти R-сервера и позволяет минимизировать задержки при обработке ответов респондента. При этом не сохраняется согласованность системы в каждый конкретный момент времени.

Масштабирование приложения, построенного на основе данной архитектуры заключается в изменении числа R-серверов по ходу работы приложения. Запуск/остановка R-сервера может производиться в любое время, при этом не требуется остановка М-сервера и/или других R-серверов. В случае отказа R-сервера возможно продолжение работы системы в нормальном режиме. В случае отказа М-сервера система теряет согласованность до перезапуска отказавшего сервера, однако сбор ответов продолжается в штатном режиме. После перезапуска М-сервера синхронизация данных происходит автоматически.

4. Реализация программного комплекса

Приложение «KtoZa» реализовано на основе клиент-серверной архитектуры. Разработан веб-интерфейс приложения и WebAPI [10] серверной части для обеспечения возможности подключения сторонних клиентов.

1. Реализация клиентской части

Для демонстрации работы приложения создана клиентская часть, которая представляет собой веб-интерфейс пользователя. Данные загружаются динамически, что продиктовано требованием модульности, а именно необходимостью наличия возможности подключения сторонних клиентов. Логика работы клиентской части реализована на языке CoffeeScript [11] с помощью фреймворка AngularJS [12], с последующей трансляцией в код JavaScript. Данный фреймворк позволяет обеспечить реализацию в рамках паттерна MVC и упрощает работу с данными, получаемыми от сервера. Графический дизайн интерфейса выполнен с помощью элементов из CSS-библиотеки Bootstrap [13]. Ввиду особенностей архитектуры приложения реализованы два независимых интерфейса, для создателя опроса и для респондента.

Интерфейс создателя опроса предоставляет возможность создания опроса в интерактивном режиме, а также просмотра текущей статистики опроса: число участников опроса, число принятых ответов, число участников, выбравших конкретный вариант ответа. Для обеспечения обновления статистики в реальном времени в качестве протокола передачи статистики был выбран протокол WebSocket [14]. Для обеспечения возможности работы контроллера AngularJS с данным протоколом использован сторонний модуль angular-websocket [15]

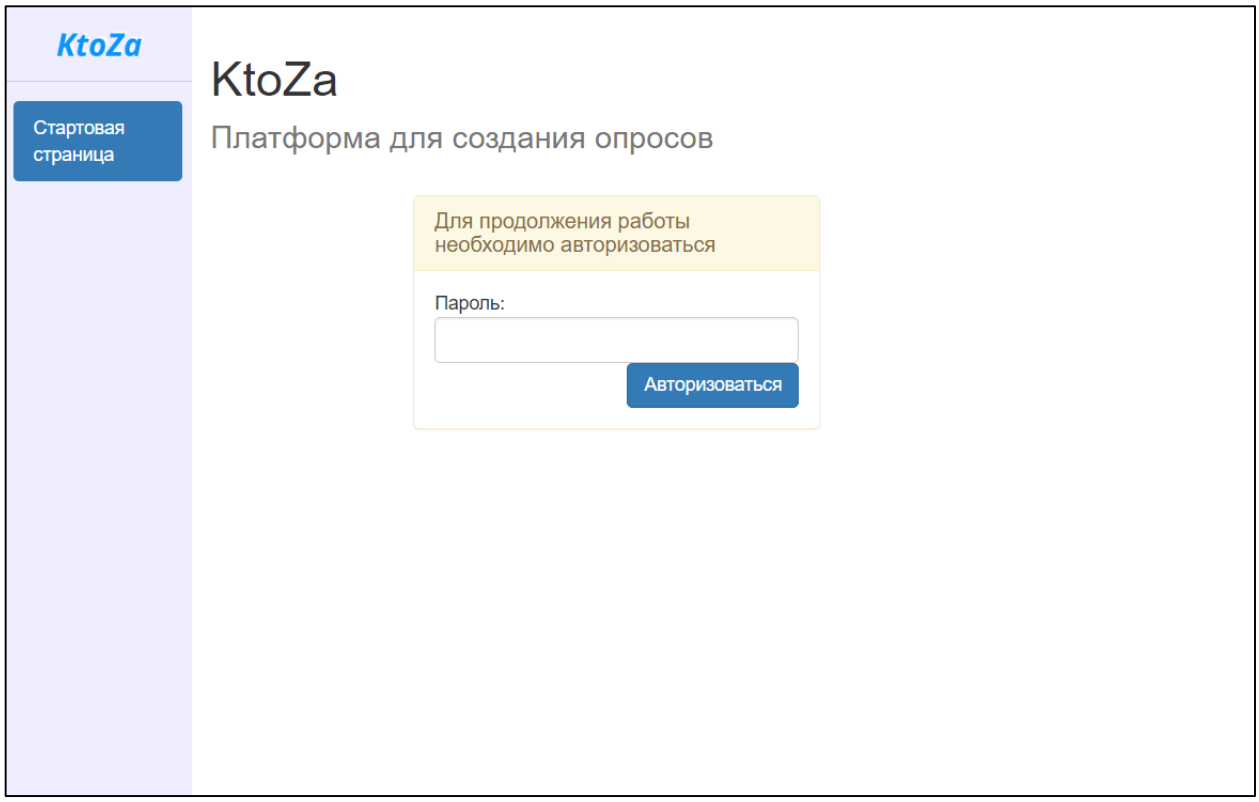

Рисунок 2. Интерфейс создателя опроса. Начало работы

В начале работы производится попытка аутентификации создателя опроса. Если клиент не аутентифицирован, отображается окно ввода пароля (рис. 2). После этого производится попытка загрузки текущих опроса и статистики. Если опроса нет, то во вкладке «Опрос» отображается приглашение к созданию опроса.

На рис. 3 представлено окно редактирования опроса создателем опроса. На данном этапе осуществляется проверка заполнения необходимых полей и правильности установленных сроков начала регистрации, начала и окончания опроса. Допускается не менее 2 вариантов ответа на каждый вопрос. Возможно добавление вопросов с одним или несколькими вариантами ответа.

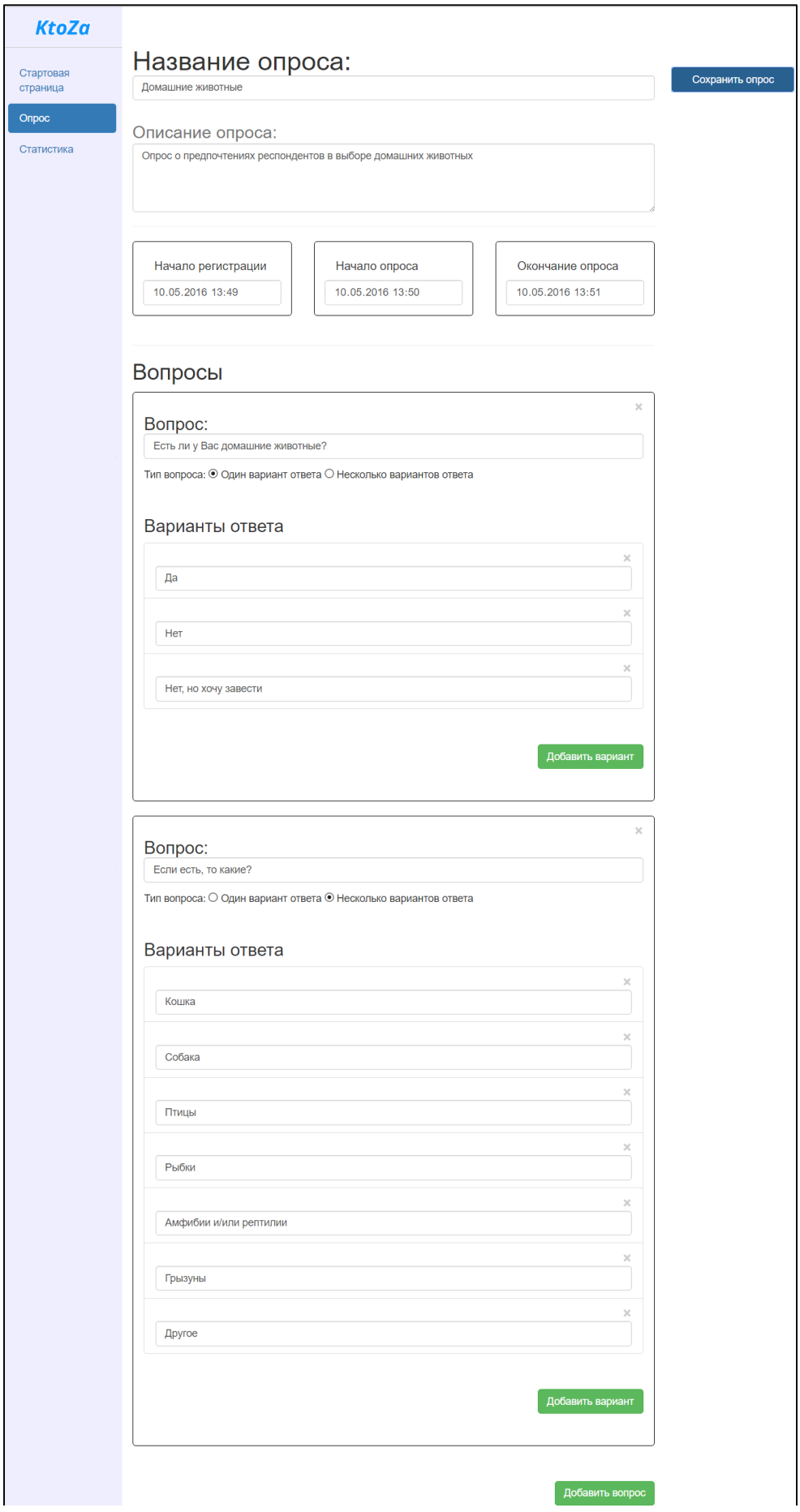

Рисунок 3. Интерфейс создателя опроса. Редактирование опроса

Интерфейс просмотра статистики для создателя опроса представлен на рис. 4. Данный интерфейс обновляется в режиме реального времени.

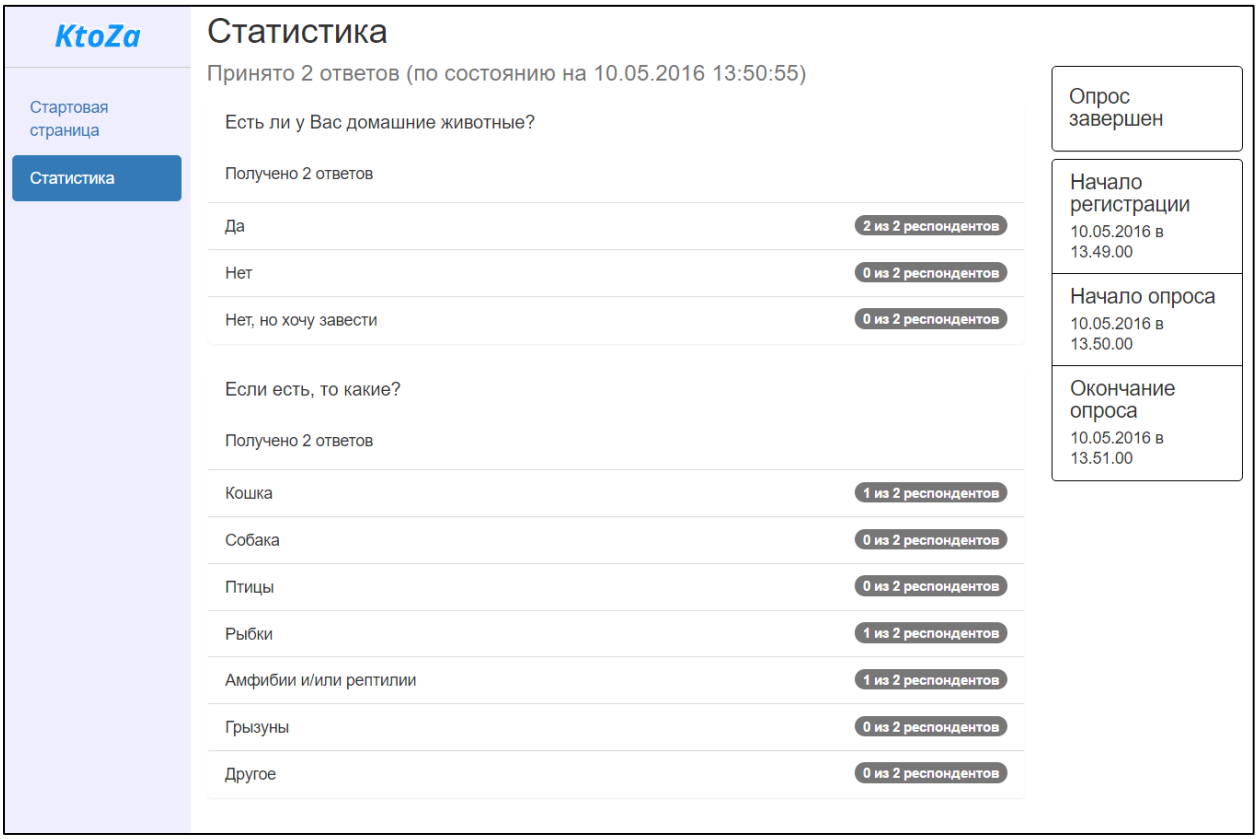

Рисунок 4. Интерфейс создателя опроса. Просмотр статистики

Клиентская часть респондента предусматривает возможность ответа на вопросы в графическом режиме. В начале работы осуществляется загрузка текущего опроса с R-сервера. Если опроса нет, то отображается соответствующее сообщение, иначе начинается обратный отсчет до начала регистрации, начала либо окончания опроса, в зависимости от текущего времени (рис.5).

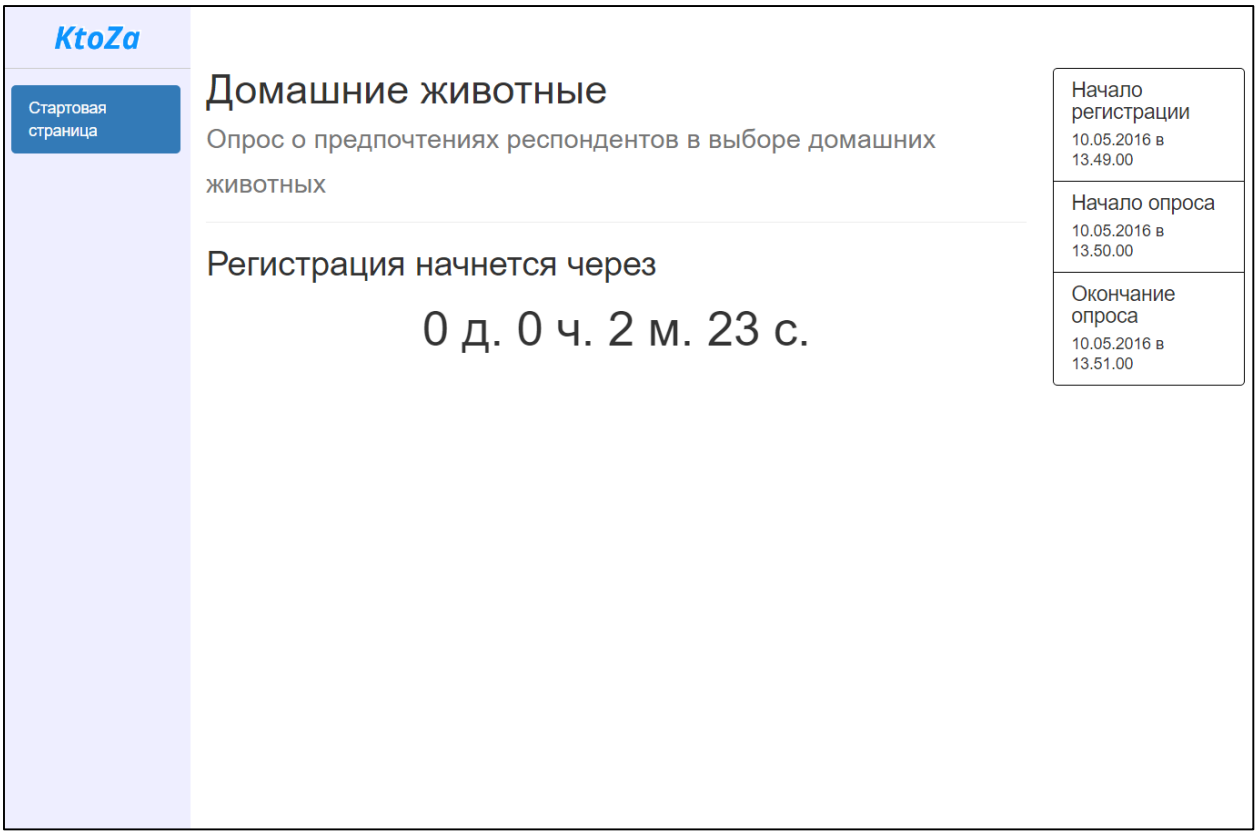

Рисунок 5. Интерфейс респондента. Стартовая страница

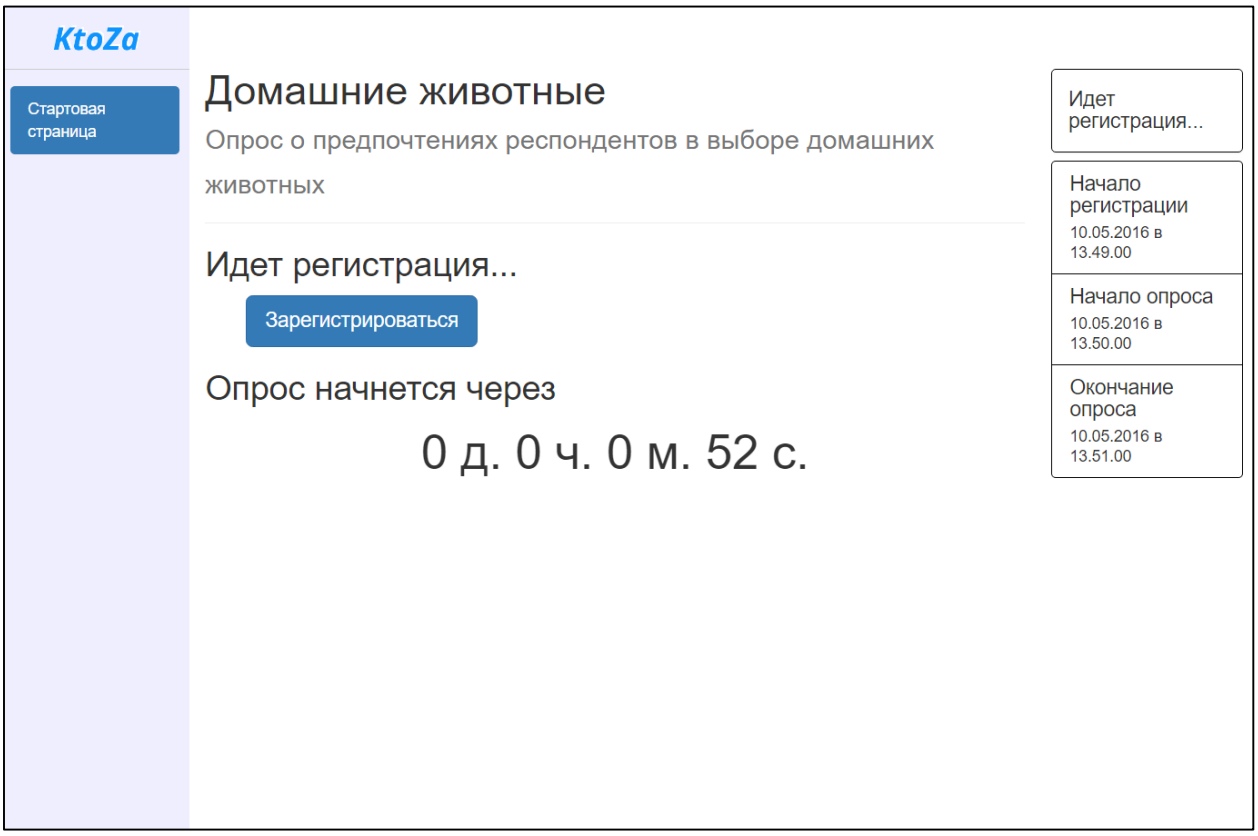

Рисунок 6. Интерфейс респондента. Начало регистрации

После начала регистрации отображается окно регистрации (рис. 6). Если по результатам проверки регистрации сервером клиент зарегистрирован, кнопка «Зарегистрироваться» становится недоступной и меняет название.

Во время опроса, если клиент зарегистрирован, становится доступной вкладка «Вопросы» (рис. 7). В ней отображается список вопросов и вариантов ответа на них. Клиентская часть осуществляет валидацию ответов и в случае, когда на вопрос получен ответ, меняет цвет вопроса на зеленый.

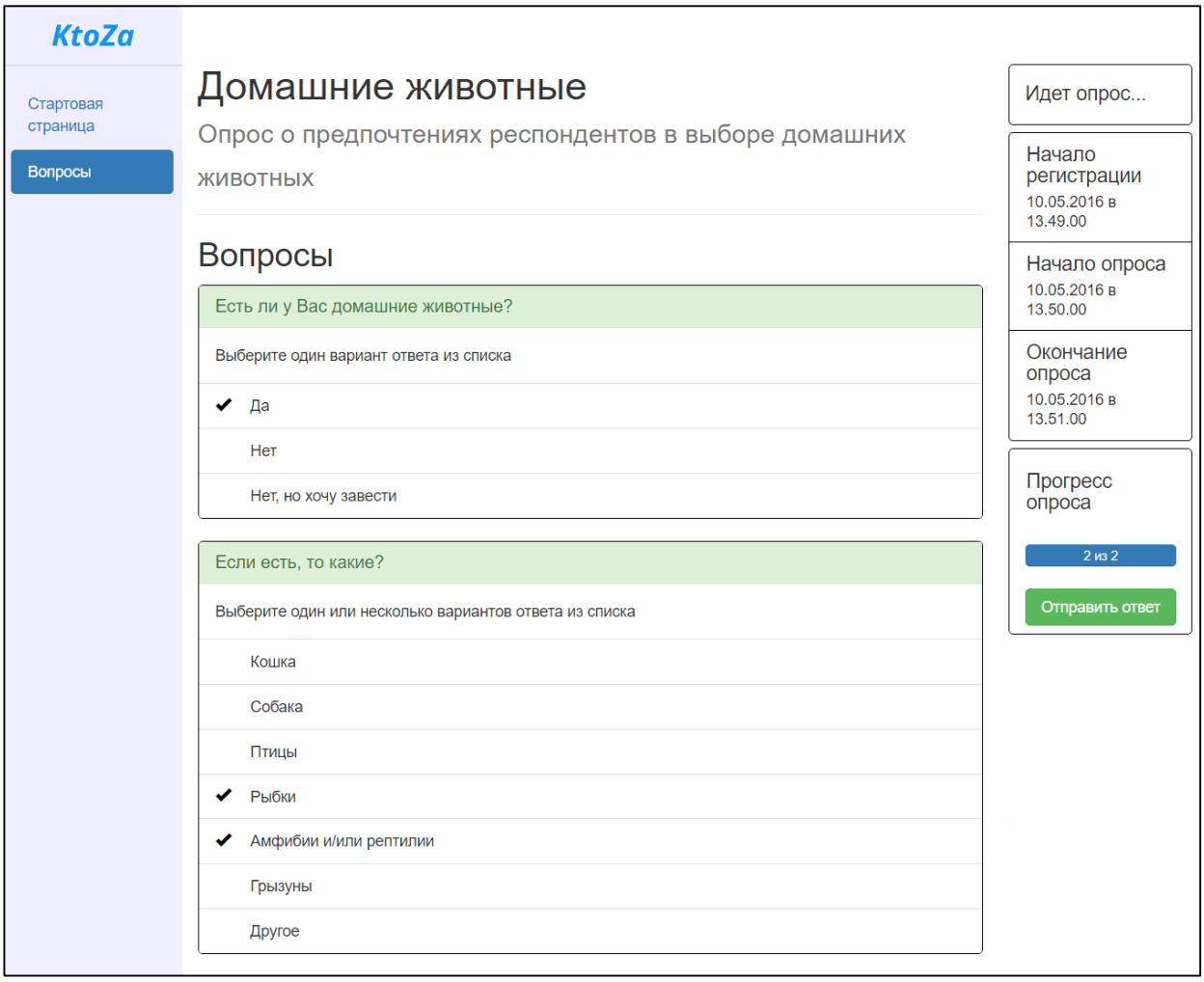

Рисунок 7. Интерфейс респондента. Процесс ответа на вопросы

После завершения опроса возможен просмотр текущей статистики (рис. 8). Однако, просмотр статистики в реальном времени не обеспечивается, и загрузка актуальных данных производится вручную.

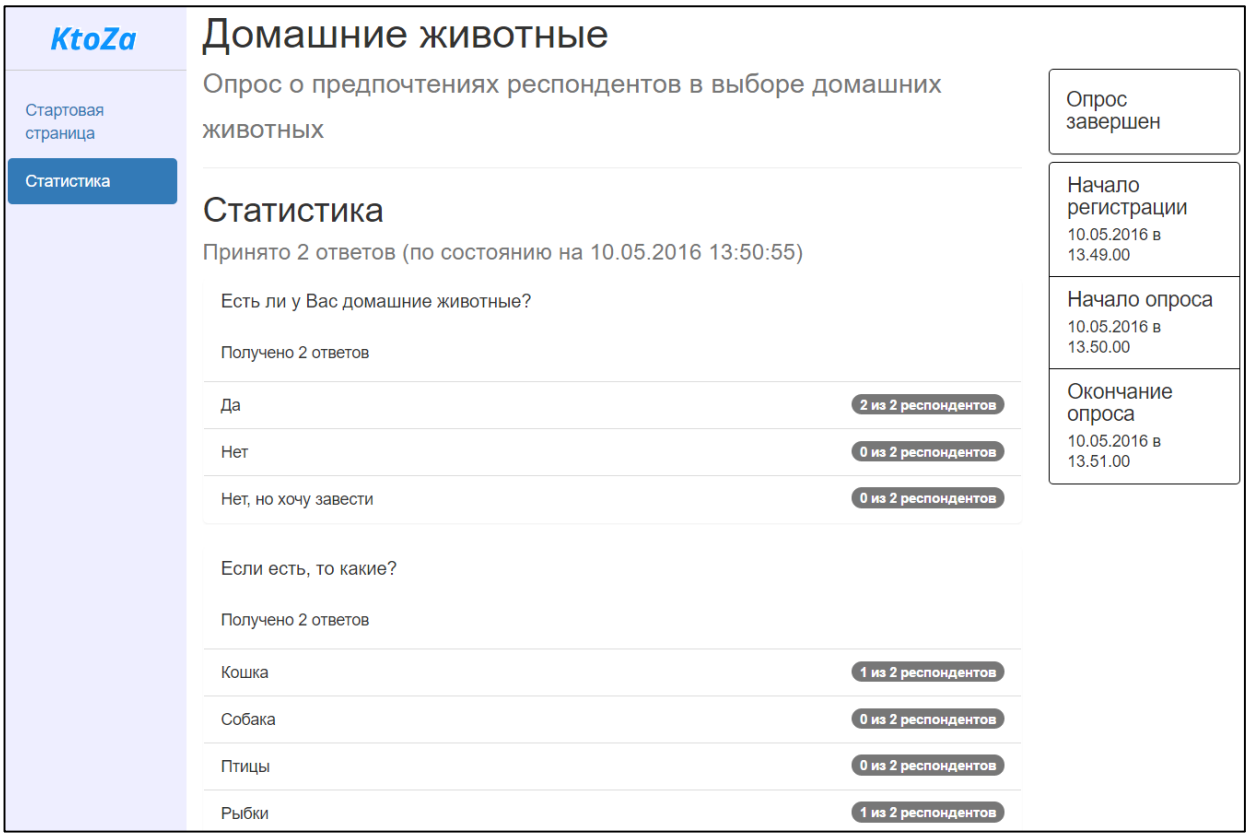

Рисунок 8. Интерфейс респондента. Просмотр статистики.

2. Реализация серверной части

Серверная часть приложения также разделена на 2 части, для R-сервера и для M-сервера. Обе части реализованы на языке Go 1.6 [5], ориентированном на разработку сетевых сервисов [16] (в т. ч. микросервисов) и многопоточное исполнение программного кода. Также обе части осуществляют отдачу статических файлов соответствующей клиентской части.

Серверная часть R-сервера реализует возможность сбора, проверки и агрегации ответов пользователя. Перед началом работы сервер получает данные опроса и снимок статистики от M-сервера. Это требует развертывание и запуск M-сервера до того, как будут запущены экземпляры R-сервера. R-сервер имеет собственный кэш ответов, представляющий собой копию статистики, в которой учтены только последние ответы. При получении нового ответа, сервер

проверяет его на соответствие типу вопроса и применяет к кэшу, при этом устанавливая флаг NeedSubmit. В отдельном потоке исполняется код синхронизатора, который при наличии флага NeedSubmit отправляет данные на M-сервер. В ответе M-сервера содержатся данные статистики **после** применения данного кэша ответов. Эти данные формируют локальную копию статистики для R-сервера. В случае запроса статистики от респондента, передается именно локальная копия статистики R-сервера.

Данный подход позволяет уменьшить задержки при обработке запросов, так как:

- локальный кэш и текущая статистика хранятся в оперативной памяти;
- синхронизация данных с M-сервером происходит асинхронно, кэш при этом не блокируется, а заменяется новой копией. Если возникает ошибка при синхронизации, то синхронизируемая копия кэша объединяется с текущей.

API R-сервера содержит следующие методы доступа:

- GET /api/poll – получение данных опроса в формате JSON
- GET /api/stats – получение данных статистики в формате JSON
- POST /api/submit – отправка ответа в формате JSON
- GET /api/register – проверка регистрации (использует cookie)
- POST /api/register – регистрация респондента

Формат данных опроса:

```
{
  "title": "Название опроса",
  "caption": "Описание опроса",
  "events": {
    "registration": "2016-05-07T14:40:00Z",
    "start": "2016-05-07T14:45:00Z",
    "end": "2016-05-07T14:50:00Z"
  },
  "questions": [
    {
       "text": "Текст вопроса",
       "type": "single-option", ["multi-option"]
       "options": ["Вариант 1","Вариант 2"]
    }
  ]
}
```

Формат данных ответа

```
[
  [0, 1], // Номера выбранных вариантов ответа на i-ый
вопрос
  [0]
]
```

Формат данных статистики / кэша ответов:

```
{
  "date":"2016-05-07T17:30:22.6557273+03:00",
  // Время последнего обновления
  "questions":[
    {
      "answerCount":0, // Число собранных ответов
      "options":[
         {
           "count":0 // Число голосов за соотв. вариант
         }
      ]
    }
  ],
  "respondents":0 // Число участников опроса
}
```

M-сервер устроен проще. Его задачами являются: обеспечение доступа к клиентской части создателя опроса, управление опросом, прием кэшей от R-серверов и добавление их к текущей статистике, сохранение статистики на диск и отправка статистики создателю опроса и R-серверам. Нагрузка на данный сервер много меньше, чем на R-сервера, что позволяет размещать M-сервер на менее производительном сервере. При получении кэша ответов сервер применяет его к текущей статистике, сохраняет статистику на диск и оправляет R-серверу актуальную статистику. При запуске M-сервера указывается пароль, запрашиваемый при заходе на страницу создателя опроса. Таким образом реализуется контроль доступа.

API M-сервера содержит следующие методы:

- GET /api/poll – получение данных опроса (JSON)
- PUT /api/poll – обновление данных опроса (JSON)
- GET /api/stats – получение данных статистики (JSON)
- GET /api/auth – проверка аутентификации
- POST /api/auth?p=<пароль> – аутентификация
- GET /api/ws – подключение к WebSocket-каналу для получения обновлений опроса и статистики

При обновлении опроса или статистики, М-сервер оповещает всех подключенных WebSocket-клиентов об обновлении и передает обновленные данные. Одновременно с этим, М-сервер прослушивает WebSocket-соединения в ожидании поступления кэша ответов. При поступлении такого кэша, М-сервер объединяет его с текущей статистикой.

5. Тестирование приложения

Одним из основных требований к приложению является отказоустойчивость. Для проверки соответствия приложения данному требованию был проведен эксперимент:

1. Запущено один экземпляр M-сервера и два экземпляра R-сервера. Соединение установлено, сервера запускаются нормально.
2. Отключен один из R-серверов. M-сервер и другой R-сервер продолжают нормальную работу. Снова запущен R-сервер.
3. Отключен M-сервер. На экран выводятся сообщения о попытках подключения к M-серверу. Прием ответов продолжается, но статистика не обновляется.
4. Снова запущен M-сервер. R-сервера успешно подключаются к M-серверу. Статистика обновляется, приняты все изменения с момента остановки M-сервера.
5. Успешное завершение эксперимента.

Другим важным требованием к приложению является производительность, особенно степень её роста при увеличении вычислительной мощности системы. Для проверки производительности были развернуты M-сервер и два экземпляра R-сервера в облачной среде DigitalOcean [17], следующей конфигурации: 1 ядро ЦП, 512 МБ ОЗУ. Составлена следующая программа проведения тестирования:

Для создателя опроса:

- 2 попытки проверки аутентификации
- 1 попытка аутентификации
- 1 запрос опроса
- 1 запрос статистики

Для респондента:

- 2 запроса опроса
- 2 запроса статистики

- 2 проверки регистрации
- 1 попытка регистрации
- 1 попытка отправки запроса

Тестирование проводилось с помощью утилиты Apache jMeter. Для тестирования подготовлены версии программы с отключенными проверками времени запроса и регистрации (проверки выполняются, но их результат не влияет на успех выполнения запроса). Программа тестирования выполняется для следующих случаев:

- число одновременных подключений: 1 создатель опроса и 100 респондентов, 5 создателей опроса и 500 респондентов, по 10 циклов выполнения программы тестирования на пользователя;
- конфигурация системы: 1 M-сервер и 1 R-сервер, 1 M-сервер и 2 R-сервера

Среднее время выполнения запроса (включая время установления соединения) приведено в табл. 1.

Число пользователей	1 созд. / 100 респ.		5 созд. / 500 респ.	
Конфигурация	1 M / 1 R	1 M / 2 R	1 M / 1 R	1 M / 2 R
M-сервер	Среднее время выполнения запроса, мс.			
GET /api/auth	107	99	563	851
POST /api/auth	140	97	578	417
GET /api/poll	96	122	451	663
GET /api/stat	96	151	741	824
R-сервер	Среднее время выполнения запроса, мс.			
GET /api/poll	147	162	1309	963
GET /api/stat	147	149	1189	1006
GET /api/register	144	153	1087	826
POST /api/register	145	159	1051	847
POST /api/submit	153	158	1112	916

Таблица 1. Среднее время выполнения запроса

Число пользователей	1 созд. / 100 респ.		5 созд. / 500 респ.	
Конфигурация	1 M / 1 R	1 M / 2 R	1 M / 1 R	1 M / 2 R
Число запросов в секунду.	497,0	464,4	118,1	396,6

Таблица 2. Производительность системы в целом

Производительность системы в целом представлена в табл. 2. Эти результаты отражают средние показатели производительности для случайно выбранного пользователя. Как видно из данной таблицы, для малой нагрузки увеличение числа R-серверов снижает производительность системы, поэтому наращивание мощностей имеет смысл только при достаточной нагрузке. Вместе с тем, при нагрузке в 5 создателей опроса и 500 респондентов, производительность выросла на 235,8%.

Отдельно протестирована скорость обновления статистики M-сервера. Было создано WebSocket-соединение с M-сервером. Одновременно производились запросы к R-серверам на добавление ответа. Измерялся промежуток времени между получениями сообщения через WebSocket. Умножая данный промежуток на число R-серверов можно оценить период обновления статистики. Результаты тестирования приведены в табл.3. Наблюдается ожидаемая обратная пропорциональность промежутка между сообщениями числу R-серверов, таким образом период обновления примерно одинаков для данных конфигураций и составляет ~15 мс. Это позволяет оценить риски, связанные с отказом R-сервера. Полагая, что система обрабатывает около 400 запросов в секунду, нетрудно подсчитать, что в течение 15 мс. будет принято около 6 запросов. Таким образом, потери от отказа R-сервера, при наличии не синхронизированных данных, составляют порядка 6 голосов. Стоит также отметить, что во время проведения тестирования случаев отказа R- или M-сервера не наблюдалось, что говорит о высокой степени отказоустойчивости.

Число пользователей	1 созд. / 100 респ.		5 созд. / 500 респ.	
Конфигурация	1 M / 1 R	1 M / 2 R	1 M / 1 R	1 M / 2 R
Промежуток между получениями сообщений, мс.	13	7	15	8
Период обновления, мс.	13	14	15	16

Таблица 3. Тестирование скорости обновления статистики

Заключение

В данной работе были рассмотрены основные принципы построения масштабируемых приложений. Также реализовано приложение опросов «KtoZa», отвечающее заданным требованиям к масштабируемости, отказоустойчивости и производительности. Разработка данного приложения основана на использовании современных методов построения высоконагруженных веб-приложений.

Для данного приложения разработана архитектура, наиболее точно соответствующая специфике работы приложения. Реализована клиентская часть приложения на основе AngularJS, с учетом принципов построения одностраничных приложений. Создана серверная часть, разделяющая функциональность между M- и R- сервером, с учетом различий в нагрузке между этими серверами. Разработка серверной части осуществлена с помощью средств языка Go, ориентированного на построение веб-сервисов и многопоточную обработку данных. Взаимодействие частей приложения осуществляется по протоколам HTTP 1.1 и WebSocket. Возможна компиляция и запуск приложения как на серверах с операционной системой Windows, так и на Linux-серверах. Для серверной части реализован WebAPI, что позволяет использовать сторонние клиенты приложения. Объем реализации составил около 3000 строк кода, из них около 2000 — серверная часть приложения. Исходный код программы доступен по адресу: https://github.com/VeselovAlex/KtoZa .

Проведено тестирование отказоустойчивости и производительности приложения в различных конфигурациях запуска и при различной нагрузке. Результаты тестирования соответствуют ожидаемым и демонстрируют выполнение требований по производительности и отказоустойчивости.

Список литературы

1. International Telecommunication Union. Overview of the Internet of things // Internet of Things Global Standards Initiative. 2012. URL: http://handle.itu.int/11.1002/1000/11559 (дата обращения: 06.05.2016).
2. Fowler M., Lewis J. Microservices // Martin Fowler's blog. 2014. URL: http://martinfowler.com/articles/microservices.html (дата обращения: 28.03.2016).
3. Wagner T. Microservices without the Servers // AWS Compute Blog. 2015. URL: http://aws.amazon.com/ru/blogs/compute/microservices-without-the-servers/ (дата обращения: 11.04.2016).
4. Coskun M. Introduction to Reliable Collections in Azure Service Fabric stateful services // Microsoft Azure. 2016. URL: https://azure.microsoft.com/en-in/documentation/articles/service-fabric-reliable-services-reliable-collections/ (дата обращения: 05.04.2016).
5. The Go Programming Language [Электронный ресурс] // The Go Programming Language: [сайт]. [2016]. URL: https://golang.org (дата обращения: 27.04.2016).
6. Hoare C.A.R. Communicating sequential processes. Springer, 1978.
7. Gerrand A. Share Memory By Communicating // The Go Blog. 2010. URL: http://blog.golang.org/share-memory-by-communicating (дата обращения: 12.04.2014).
8. Ньюмен С. Создание микросервисов. Спб.: Питер, 2016. 304 с.
9. Greiner R. CAP Theorem: Revisited // Robert Greiner Blog. 2014. URL: http://robertgreiner.com/2014/08/cap-theorem-revisited/ (дата обращения: 27.04.2016).
10. Richardson L., Amundsen M. RESTful Web APIs. CA, Sebastopol: O'Reilly, 2013. 404 pp.
11. Бейтс М. CoffeeScript. Второе дыхание JavaScript. М.: ДМК Пресс, 2012. 312 с.

12. AngularJS — Superheroic JavaScript MVW Framework [Электронный ресурс] // AngularJS: [сайт]. [2016]. URL: https://angularjs.org (дата обращения: 27.04.2016).
13. About - Bootstrap [Электронный ресурс] // Bootstrap - The world's most popular mobile-first and responsive front-end framework.: [сайт]. [2016]. URL: http://getbootstrap.com/about (дата обращения: 27.04.2016).
14. IETF. The WebSocket Protocol // IETF Tools. 2011. URL: https:// tools.ietf.org/html/rfc6455 (дата обращения: 27.04.2016).
15. Angular Websocket [Электронный ресурс] // GitHub: [сайт]. [2016]. URL: https://github.com/AngularClass/angular-websocket (дата обращения: 06.05.2016).
16. Donovan A., Kernighan B. The Go Proramming Language. Addison-Wesley, 2016. 400 pp.
17. Simple Cloud Computing, Built for Developers. [Электронный ресурс] // DigitalOcean: [сайт]. [2016]. URL: https://www.digitalocean.com (дата обращения: 07.05.2016).